SigTest Tool 2.0 User's Guide

For Java™ Compatibility Test Suite Developers

Sun Microsystems, Inc.
www.sun.com

April 2008

Submit comments about this document at: `http://java.sun.com/docs/forms/sendusmail.html`

Contents

Tables

Code Examples

Preface

This guide describes how to install and run the SigTest tool. This tool is composed of a group of utilities used to develop signature test components that can be used to compare API test signatures.

Note – For simplicity, this user's guide refers to the test harness as the *JavaTest harness*. Note that the open source version of the harness, called *JT harness*, can be used in its place. The JT harness software can be downloaded from: `http://jtharness.dev.java.net/`

Who Should Use This Guide

This guide is for developers of quality assurance test suites and developers of compatibility test suites — TCKs for a Java™ platform API as part of the Java Community Process™ (JCP™) program.

Before You Read This Guide

Before reading this guide, it is best to be familiar with the Java programming language. A good resource for the Java programming language is the Sun Microsystems, Inc. web site, located at `http://java.sun.com`.

Note – Web URLs provided are subject to change.

How This Guide Is Organized

Chapter 1 describes SigTest tool and the purpose of signature testing.

Chapter 2" provides a synopsis of each of the SigTest tool commands along with their available options and arguments.

Appendix " provides SigTest tool command examples that you can run to quickly familiarize yourself with them.

Related Documentation

For details on the Java programming language, see the following documents:

- *The Java Programming Language, Third Edition*
- *The Java Language Specification, Second Edition*
- *The Java Virtual Machine Specification, Second Edition*

These documents are available at `http://java.sun.com/docs/books/`.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories, or on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable or placeholder. Replace with a real name or value	To delete a file, type `rm` *filename*. *SigTest-Directory*[*]

\ or ^	A backslash at the end of a line indicates that a long code line has been broken in two on a UNIX® system, typically to improve legibility in code. The caret character (^) indicates this on a Microsoft Windows system.	`java classname \`
		`[`*classname_arguments*`]`
		
		`java classname ^`
		`[`*classname_arguments*`]`

Indented code or command line	Indicates a wrapped continuation from a previous line with no carriage return or return character in the actual code.	java classname `[`*classname_arguments*`]`

[*] The top-most SigTest tool installation directory is referred to as *SigTest-Directory* throughout the SigTest tool documentation.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Send us your comments at `http://java.sun.com/docs/forms/sendusmail.html`.

CHAPTER 1

Introduction

The SigTest tool makes it possible to easily compare the signatures of two different implementations of the same API. It verifies that all of the members are present, reports when new members are added, and checks the specified behavior of each API member.

Signature Test Basics

A signature test compares two implementations of an API and reports the differences. SigTest tool compares the signatures of two implementations of the same API and and can do the following:

- Create and run a test that verifies that all of the members are present
- Report when new members are added
- Check the specified behavior of each API member

The signature test created by SigTest tool can be run independently at the command line, or under the control of the JavaTest™ harness.

Note – For simplicity, this user's guide refers to the test harness as the *JavaTest harness*. Note that the open source version of the harness, called *JT harness*, can be used in its place. The JT harness software can be downloaded from:
`http://jtharness.dev.java.net/`

The SigTest tool was originally created to assist in the creation of Java technology compatibility test suites (TCKs). It simplified the process of verifying that the API signature of a new implementation of a Java technology matched the signature of a reference implementation of that API.

When used in a software development environment, SigTest tool can be used to track and control changes to an API throughout the development process.

What is Tested

The signature test algorithm compares the API implementation under test with a signature file created from the API you are comparing it to — often referred to as a *reference implementation*. The signature test checks for mutual binary or source compatibility by verifying the equality of API member sets. By checking for equality of API member sets, the test verifies that the following conditions are true:

- If an API item is defined in the reference representation of the API, then that item is implemented in the API under test, and vice versa.
- Attributes chosen for comparison are identical in both implementations of the API. The tool chooses attributes for comparison according to the type of check being processed. This is described more in the next section.

Mutual Binary Compatibility Check

The signature test binary compatibility check mode verifies that a Java technology implementation undergoing compatibility testing and its referenced APIs are mutually binary compatible as defined in Chapter 13, “Binary Compatibility,” of *The Java Language Specification*. This assures that any application runs with any compatible API without any linkage errors.

This check is less strict than the default source compatibility check, described next. It is for use primarily in the special case of when a technology is developed for Java technology environments that are purely runtime. Such an environment does not provide a Java technology-based compiler (Java compiler), nor does it include class files that could be used to compile applications for that environment. Because of the limited use of such an environment, the API requirements are slightly relaxed compared to environments that support application development.

Java application environments can contain several Java technologies. Not all Java technologies can be combined with each other, and in particular, their sets of API signatures might be incompatible with each other. Relaxing signature checks to the level of mutual binary compatibility allows the developer to combine technologies in a purely runtime environment that cannot be combined otherwise.

Mutual Source Compatibility Check

While binary compatibility is important, it cannot guarantee that an application in binary form as a set of class files can be *recompiled* without error.

The signature test source compatibility check mode verifies that any application that compiles without error with a compatible API, compiles without error with all other source compatible APIs.

Mutual source compatibility is a stricter check than the mutual binary compatibility and SigTest tool performs it by default.

Class and Class Member Attributes Checked

A Java platform API consists of classes, and interfaces, and their member fields, methods, and constructors, and documented annotations. In turn, all of these API items can have various attributes such as names, modifiers, a list of parameters, a list of interfaces, exceptions, nested classes, and so forth. A signature test checks that certain members and attributes belonging to the API under test are the same as those defined by the API to which it is being compared. The signature test checks public and protected API items only and ignores private and package-access items.

The tool checks the following attributes when comparing items in the API implementation under test:

- Classes and interfaces, including nested classes and interfaces:
  - Set of modifiers except `strictfp`
  - Name of the superclass
  - Names of all superinterfaces, direct plus indirect, where order is insignificant
- Constructors:
  - Set of modifiers
  - List of argument types
  - In source compatibility mode only, the normalized list of thrown exceptions where order is insignificant

    Normalizing the throw lists involves removing all superfluous exception classes. An exception class is superfluous if it is a subclass of either the `java.lang.RuntimeException` class, the `java.lang.Error` class, or another class from the same list.
- Methods:
  - The set of modifiers, except `strictfp`, `synchronized`, and `native`
  - The return type
  - The list of argument types

- In source mode only, the normalized list of thrown exceptions, described earlier, where order is insignificant
- Fields:
  - Set of modifiers, except `transient`
  - Field type
- Documented annotations with `SOURCE` and `RUNTIME` retention of the following types:
  - Classes and interfaces
  - Fields, methods and constructors
  - Parameters and annotation types

The tool performs the check in the following order:

1. For all top-level public and protected classes and interfaces, it compares the attributes of any classes and interfaces with the same fully qualified name.
2. Taking into account all declared and inherited members, it compares all public and protected members of the same kind and same simple name, treating constructors as class members for convenience sake.

Source and Binary Compatibility Modes

Earlier SigTest tool versions performed a comparison of all exceptions declared in `throws` clauses for methods and constructors. Certain variations in this area caused an error message during the signature test. Despite these error messages, the source files compiled successfully together. Successful compilation is the basic criteria for source compatibility with the the current SigTest tool, while successful linking as the basic criteria for binary compatibility.

Changes to the `throws` clause of methods or constructors do not break compatibility with existing binaries because these clauses are checked only at compile time, causing no linkage error. For the purpose of signature testing, this relates directly to binary compatibility as described earlier in "Mutual Binary Compatibility Check" on page 2.

The adaptation of JSR 68, *The Java™ ME Platform Specification*, formalized the use of building blocks in API development. A building block is a subset of an existing API that is approved for reuse in the construction of profiles or optional packages. The building block concept enables a developer to duplicate the functionality provided by another API without having to redefine an entirely new API. For further details see JSR 68 at
`http://www.jcp.org/en/jsr/detail?id=68`.

The use of building blocks created a need for more lenient checking of exception throw lists compared to earlier SigTest tool versions. Consequently, SigTest tool 1.5 provides both a source and a binary compatibility mode of operation. This retains compatibility with earlier signature files while adding support for building blocks and eliminating the unnecessary error messages.

The `SignatureTest` command recognizes the `-mode` option that takes the values "src" or "`bin`" as arguments for choosing source mode or binary mode. The choice of which mode to use depends on the type of signature file being used in the test. This is described in more detail later in these sections:

- "`Setup` Command" on page 14 describes how to generate a signature file
- "`SignatureTest` Command" on page 20 describes how to specify the mode when running a signature test
- "`Merge` Command" on page 27 describes how to generate a combined signature file from set of signature files

The difference between the binary and source compatibility modes is how the tool handles the `throws` list for constructors and methods (as described in "Class and Class Member Attributes Checked" on page 3). Constant checking behavior is also different in binary and source compatibility modes. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. "Constant Checking in Differing Run Modes" on page 9 describes these differences in more detail.

Using Custom Signature Loaders

The signature test has a requirement for the Java™ Platform, Standard Edition (Java SE platform) runtime environment version 1.4 or later. This requirement might prevent use of the tool on limited or nonstandard environments such as some Java™ Platform, Micro Edition (Java ME platform) or Java™ Platform, Enterprise Edition (Java EE platform) configurations.

To overcome this, the tool provides support for custom signature loaders that can be implemented as plug-ins. These plug-ins gather signatures from a runtime environment when the `SignatureTest` command cannot be run directly. For example, you might create a light-weight remote JavaTest harness agent and run the signature loader on a remote Connected Device Configuration (CDC) compatible device. Another example is using a wrapped J2EE platform bean as a signature loader inside a J2EE platform container where any direct file I/O operations are prohibited.

As an aid in developing such an extension, the SigTest tool distribution includes a class library that contains a signature serializer and some related utility classes in the *SigTest-Directory*`/lib/remote.jar` file. This file contains a subset of the SigTest tool classes that are necessary to develop a custom plug-in. All of these library classes are CDC 1.0 compatible and have minimal memory requirements. The source code for these classes is distributed in the *SigTest-Directory*`/redistributables/sigtest_src.zip` file. The code is designed for running a plug-in with the JavaTest harness using the Java ME Framework. The server and client source code and the HTML test descriptions for an actual plug-in example are located in the *SigTest-Directory*`/examples/remote` directory.

Note – The open source version of the ME Framework is available at: `http://cqme.dev.java.net/framework.html`.

CHAPTER 2

Using the Signature Test Tool

This chapter provides a synopsis of each of the SigTest tool commands along with their available options and arguments. It contains these sections:

Also see Appendix A for examples of each SigTest tool command that you can run.

Signature Test Tool Basics

The SigTest tool operates from the command line to generate or manipulate signature files. A signature file is a text representation of the set of public and protected features provided by an API. Test suite developers include it in a finished test suite as a signature reference for comparison to the technology implementation under test. The following list shows the commands that are available.

- `Setup` - Creates a signature file from either an API defined by a specification or a reference API implementation.
- `SignatureTest` - Compares the reference API represented in the signature file to the API under test and produces a report. This is the test that becomes part of a finished test suite.
- `SetupAndTest` - Executes the `Setup` and `SignatureTest` commands in one operation.

- `Merge` - Creates a combined signature file from several signature files representing different Java APIs in one Java runtime environment according to the JSR 68 rules.

The SigTest tool distribution includes a Java™ Archive (JAR) file used for developing a signature test and one for distribution within a finished test suite to run its signature test. The description of each follows:

- `sigtestdev.jar` - Contains classes for running the commands used during signature test development.
- `sigtest.jar` - Contains only the classes for running the `SignatureTest` command. This file is distributed in a finished test suite.

Test suite developers perform these operations while using `sigtestdev.jar` to develop a signature test.

1. Run the `Setup` command to create a signature file from either an API defined by a specification or a reference API implementation.

2. Include the files required to run the signature test in the finished test suite distribution.

Testers perform these operations using `sigtest.jar` while testing the implementation, as described in "Running a Signature Test With the JavaTest Harness" on page 23.

1. Run the `SignatureTest` command to compare the reference representation of the API in the signature file to the API under test and produce a report. This step is usually performed under the control of the JavaTest harness as described in "Running a Signature Test With the JavaTest Harness" on page 23.

2. Interpret the report.

Reflection and Static Run Modes

Two run modes are available during command execution. These modes determine how the class descriptions are examined and retrieved, as follows:

- **Reflection Mode -** Uses reflection to examine API classes and retrieve information about them. The reflection mode is of greatest advantage when the API to be analyzed has no external class files.
- **Static Mode -** Specified with the `-static` flag, the tool parses only the class files listed in the `-classpath` command-line argument.

Note – In static mode you can test specified classes in another runtime environment. For example, this can be useful to analyze APIs that are part of a Java SE platform 1.4.2 environment when the `SignatureTest` command is run on a Java SE platform version 5.0.

Constant Checking in Differing Run Modes

The requirements related to constant checking differ in binary and source compatibility testing. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. Use the `-static` mode to enforce strict constant checking in source code compatibility testing.

When running a signature test in source compatibility mode and using the static mode, constant checking is strict and two way. This means that all the constant fields and their related values specified in the reference API must have the same values in the API under test. Likewise, all the constant fields and their related values specified in the API under test must have the same values in the reference API.

In binary compatibility mode, the requirements related to constant checking are less strict. The signature test verifies that all the constant fields and associated values contained in the reference API are also available in the API under test. If any field values are missing or different, it reports an error. However, the signature test does not report an error if constant values are found in the API under test that are not available in the reference API.

Generics Checking in Binary Mode

The information related to generics is not used by the Java Virtual Machine[1] at runtime. This information is used only by the compiler at compile time.

In binary mode the `SignatureTest` command compares the signatures of parameterized types after omitting the type parameters and arguments from both the signature file and the analyzed API (termed type erasure). This is to ensure that they are compatible at runtime. See *The Java Language Specification, Third Edition,* for a detailed description of type erasure.

The bridge methods that are generated by the compiler during type erasure are not a part of the API and so they are ignored by the SigTest tool.

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform

`CLASSPATH` and `-classpath` Settings

TABLE 2-1 lists the requirements for setting the `CLASSPATH` environment variable and the `-classpath` argument when running either the `Setup` or `SignatureTest` commands. The table uses the following terms to describe the classes that must be included:

- **Required classes -** All superclasses and superinterfaces of the classes under test
- **Classes under test -** The set of classes specified by a combination of the following options:
  - `-package`
  - `-PackageWithoutSubpackages`
  - `-exclude`

TABLE 2-1 Settings for the `Setup` and `SignatureTest` Commands

Variable or Argument	In Reflection Mode	In Static Mode
`CLASSPATH` environment variable	Must contain these: • `sigtestdev.jar` or `sigtest.jar`. • Classes under test. • Other required classes, except for bootstrap and extension classes described in "Bootstrap and Extension Classes" on page 10.	Must contain either `sigtestdev.jar` or `sigtest.jar`.
`-classpath` argument	Required. Must contain the classes under test.	Must contain these: • Classes under test. • All required classes.

Bootstrap and Extension Classes

Bootstrap and extension classes are those located in `rt.jar` and several other JAR files under the *Java-Home*`/jre/lib/` directory, where *Java-Home* is the base directory of the Java platform runtime installation. For example, classes from the `java` package such as `java.lang.Object` are bootstrap classes. Their location is already available to the Java Virtual Machine environment. Because of this, they do not need to be specified in the `CLASSPATH` environment variable when reflection mode is used. Furthermore, bootstrap and extension classes are always loaded from JAR files located in the *Java-Home*`/jre/lib/` directory, regardless of whether they were specified in the class path or not. This is an important feature of the reflection mode.

Signature File `Merge` Rules

The `-Files` argument of the `SignatureTest` command accepts values to specify one or more signature files that are combined to represent an API configuration that is used as input for testing. This resulting API-set can also be combined into a single signature file for testing purposes.

By default the API combination is assumed to be constructed according to the JSR 68 rules. This can be overridden during a SigTest tool test run by specifying the `-NoMerge` option to run the signature test. The `-NoMerge` option forces SigTest tool to use the first class description it encounters if more than one class with the same name is found in the set of signature files specified by the `-Files` argument. In this case it uses the class from the left-most signature file that is specified with `-Files`.

JSR 68-Based Merge

The Java ME platform architecture specified by JSR 68[2] allows for the inclusion of several Java platform APIs in one conforming Java platform runtime environment. The condition for combining these APIs is that any application written for the resulting runtime environment must execute successfully on the combination.

If such a combination exists, it is defined on the basis of the combined sets of APIs with semantics, and the semantics must be compatible with those of all the included components.

The only means of verifying the semantics of the combination is to run the applicable test suite for each API. However, it is possible to automate the creation of a combined set of API signatures, and it is also possible to detect when a combined set cannot be built.

The `Merge` command functionality combines (merges) several input signature files into one resulting signature file, as follows: If A, B, and C are signature files, then A + B yields signature file C, and each of the three signature files represent the corresponding classes of their respective APIs. The `Merge` process constructs the resulting API C out of the two input APIs A and B.

`Merge` Command Operative Principles

The `Merge` command operates according to the following principles, where A and B are input APIs that are combined into the resulting API C:

2. `http://jcp.org/en/jsr/detail?id=68`

- The `Merge` operation is commutative, so with API A and B, A + B = B + A.
- It recognizes either binary or source compatibility when merging APIs.
- For any application X that is compatible with either API A or B, when A and B are merged then X must be compatible with the resulting API C.
- The resulting API C cannot contain a class that is not found in either of the A or B input APIs. This means that any class in C has to have corresponding classes in either A or B or both A and B.
- API C must not contain a class member that is not found in its corresponding classes in A and B. This applies only to declared class members and not inherited members.
- If some class in A or B, or both, has a member that overrides a member from a superclass, then the corresponding class in C must also have this overriding member.
- Each API element in C has a set of attributes derived from the attributes of its corresponding elements in A and B, and this is the smallest possible set of attributes that does not break compatibility. So if `attr` is an attribute of an element from API C, then `attr` must be defined for the corresponding element from A or B, and `attr` can not be omitted without breaking compatibility between A and C or between B and C.
- No unnecessary APIs or relationships between classes or interfaces can be introduced.

The basic algorithmic rules for combining two input APIs A and B into a signature file that represents the resulting API C are as follows:

- If one of the input APIs A or B contains an element that the other does not, then this element goes into the resulting signature file of API C without modification except for the following case: If the element in question is the first declared class member in the inheritance chain of input API A or B, and the other input API inherits the same element, then this element represented the resulting API C.
- If both of the input APIs contain two identical elements, only one of them is represented in the resulting API.
- If both of the input APIs contain a corresponding element, but with a different set of attributes, then either of the following occurs:
  - A conflict wherein the resulting API can not exist.
  - A compromise wherein the new element with a composite set of attributes is created and it is represented in the resulting API-set.

Element Handling by `Merge`

General rules for handling elements of all kinds during the `Merge` process are as follows.

- When there are two different access modifiers select the more visible one.

  For example, if A is a `public int foo`, and B is `protected int foo`, then the merge into C results into `public int foo`.
- If the elements differ in the `final` modifier, do not include it If a class is `final`, then all of its methods are implicitly `final` according to Section 8.4.3.3 of *The Java Language Specification, 2nd Editon*.
- If corresponding elements differ in the `static` modifier, then declare a conflict.

Element-specific rules are as follows:

- If corresponding classes differ in the `abstract` modifier, then declare a conflict.
- Apply the following rules for classes or interfaces and nested classes or interfaces, where for the purpose of this description, c1 and c2 are corresponding classes from the input APIs:

  If a superclass of c1 is a subclass of a superclass of c2, use the superclass of c1 as the superclass for the new element. Otherwise, if a superclass of c2 is a subclass of a superclass of c1, use the superclass of c2 as the superclass for the new element. If neither of the previous two conditions are possible, then declare a conflict.
- For classes or interfaces and nested classes or interfaces, create a combined set of superinterfaces of the corresponding classes and dismiss duplicates. Use the combined set for the new element.
- For methods and constructors, construct a `throws` list as follows:
  - In binary compatibility mode, an empty `throws` list results independently of the source lists.
  - In source compatibility mode, both `throws` lists are normalized as described in TABLE 2-3 before they are compared. If the normalized lists are equal, one is used as the result, otherwise, a conflict is declared.
- Methods that differ in the `abstract` modifier are not included.
- If a method result type is not identical a conflict is declared.
- If a field value type is not identical a conflict is declared.
- If a field element differs in the `volatile` modifier, it is included.
- Process `constant` field values as follows:
  - If one of the fields has a constant value and other does not, include the constant value in the result field.
  - If both fields have a constant value then declare a conflict if the values are different, otherwise include the value in the result field.

`Setup` Command

The `Setup` command has the following synopsis:

```
java com.sun.tdk.signaturetest.Setup [arguments]
```

TABLE 2-2 describes the available command arguments and the values that they accept. Before running the command, also see these sections: "`CLASSPATH` and `-classpath` Settings" on page 10 and "Case Sensitivity of Command Arguments" on page 16.

Command Description

The `Setup` command accepts a reference implementation of an API as input. The command processes the API input to generate a signature file that represents the API to be used as a reference of comparison for the purpose of signature testing.

`Setup` processes the API input in the static mode by parsing the set of classes specified with the `-classpath` arguments.

Also see Appendix A for an example of the command that you can run.

TABLE 2-2 `Setup` Command Arguments

`Setup` **Option**	**Description**
`-help`	Optional. Displays usage information for available command arguments and exits.
`-debug`	Optional. Enables printing of the stack trace for debugging purposes if `Setup` fails.
`-static`	Required. Specifies to run in static mode.
`-classpath` *path*	Required. Specifies the path to one or more APIs that generate the signature file. Can contain multiple directories or ZIP or JAR files. The `-package` argument further refines the set of classes specified in `-classpath` (see "`CLASSPATH` and `-classpath` Settings" on page 10). There is no default `-classpath`. Use the path separator appropriate for the platform (identified by `java.io.File.pathSeparator`).

TABLE 2-2 `Setup` Command Arguments *(Continued)*

`Setup` **Option**	**Description**
`-TestURL` *path*	Optional. Specifies the directory location in which to create the signature file as a `file` protocol URL: `file://`*path* Must end in a trailing slash on a UNIX system or a backslash on a Microsoft Windows or DOS system. `Setup` does not support the HTTP protocol.
`-FileName` *file_name*	Required. Specifies the name of the signature file to be created.
`-ClosedFile`	Optional. The default if not specified. Specifies to include in the signature file all direct and indirect superclasses for all required classes (tested classes), even if these superclasses are non-public or from untested packages.
`-NonClosedFile`	Optional. Declines the default `-ClosedFile` mode previously described. Does not include all direct and indirect superclasses and superinterfaces of tested classes in the signature file
`-package` *package_or_class_name*	Optional. Specifies a class or package to be included in the signature file, including its subpackages if a package is specified. The `-package` value acts as a filter on the set of classes specified in `-classpath`. The default is all classes. Repeat the argument to specify multiple entries.
`-PackageWithoutSubpackages` *package*	Optional. Similar to the `-package` option, this specifies a package to be included but without its subpackages. Repeat the option to specify multiple entries.
`-exclude` *package_or_class_name*	Optional. Specifies a package or class to be excluded from the signature file, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-verbose`	Optional. Enables error diagnostics for inherited class members.
`-apiVersion` *version_string*	Optional. Specifies the API version string to be recorded in the second line of the signature file, as described in "Signature File Contents" on page 17.

Case Sensitivity of Command Arguments

The specification of each argument flag at the command line is not case sensitive, but the input value entered immediately after the argument flag is case sensitive.

The following two command lines produce identical results for the `-FileName` flag:

```
% java com.sun.tdk.signaturetest.Setup -FileName name.sig
```

```
% java com.sun.tdk.signaturetest.Setup -filename name.sig
```

However, these two might not produce identical results if the host operating system is case sensitive to the file name values entered:

```
% java com.sun.tdk.signaturetest.Setup -FileName name.sig
```

```
% java com.sun.tdk.signaturetest.Setup -FileName NAME.sig
```

Signature File Formats

The SigTest tool has changed signature file formats through progressive versions. TABLE 2-3 lists the existing signature file formats and describes how each relates to a specific SigTest tool version. In SigTest tool 1.5, the `SignatureTest` and `Merge` commands read v2.1 and later signature files, and output only v4.0. The v4.0 file format supports added functionality, such as generics and annotations.

TABLE 2-3 Signature File Format Compatibility

Format	**Description**
`v0`	Generates a signature file with simple class member names. This was the default format in SigTest tool 1.0, but is not supported by `SignatureTest` command in SigTest tool 1.3 and later.
`v1`	Generates a signature file with fully qualified class member names. This was the default format in SigTest tool 1.1. This format includes *non-normalized* exception throw lists for constructors and methods. Normalizing the throw list involves removing all superfluous exception classes. A class is superfluous if it is a subclass of either the `java.lang.RuntimeException` class, or the `java.lang.Error` class, or another class from the same list. This format is not supported by `SignatureTest` command in SigTest tool 1.3 and later.
`v2`	This is the default format for SigTest tool 1.2. Generates a signature file with fully qualified class member names and modified `supr` statements. This format includes *normalized* exception throw lists for constructors and methods. This format is not supported by `SignatureTest` command in SigTest tool 1.3 and later.

TABLE 2-3 Signature File Format Compatibility *(Continued)*

Format	Description
v2.1	This version extends the v2 format to indicate whether an interface is inherited directly or indirectly. It is read by `SignatureTest` command in SigTest tool 1.3 and later.
`v3.1`	Generates data for JDK software version 5.0 such as generics, annotations, and enums.
`v4.0`	Inherited members are not written to the signature file. Private and default visibility fields and nested classes that can potentially hide visible API elements are tracked. In SigTest tool 1.5, all output files are of this version.

Also see "Source and Binary Compatibility Modes" on page 4.

Signature File Contents

`Setup` generates each signature file with a mandatory header in the first two lines, followed by the body of the signature file.

Note – Comment lines start with the pound (#) character and can be inserted anywhere after the first two mandatory header lines.

Signature File Header

`Setup` generates the first two mandatory header lines of each signature file as follows:

`#Signature file` *format*

`#Version` *version-string*

With the following variable replacement values:

- *format* is either one of the values described in TABLE 2-3, or empty, that indicates v0.
- *version-string* is a value taken directly from the argument given at the startup command line to the `-apiVersion` option (see TABLE 2-2).

Signature File Body

The remaining body of a signature file immediately follows the header. It contains the following information, which is further clarified in TABLE 2-4:

- For each `public` or `protected` class, all modifiers except `strictfp`, and the fully qualified name of any superclass or interfaces implemented, generic type parameters, and annotations.
- For each `public` or `protected` interface, all modifiers except `strictfp`, and the fully qualified name of any superinterfaces implemented, generic type parameters, and annotations.
- For each `public` or `protected` field, all modifiers except `transient`, the fully qualified name of the field's type and its fully qualified name. If the field is a primitive or string constant, the value of the field is included.
- For each `public` or `protected` method, all modifiers (except `native`, `synchronized`, and `strictfp`), the fully qualified name of the type of returned value, the method's fully qualified name, types of all parameters, and the names of exceptions declared in a `throws` clause.
- For each `public` or `protected` constructor, all modifiers, the fully qualified name of the constructor, types of all parameters, and any exceptions declared in a `throws` clause.

Note – All `private types` that are used in the definition of a `public` or `protected` item are substituted by their `public` or `protected` equivalent if possible, otherwise an error is generated. All types included in a signature file are either `public` or `protected` and not `private` or `package local`.

TABLE 2-4 further summarizes the contents of a generated signature file. A plus (+) indicates a class modifier is included in a generated signature file and a minus (-) indicates it is ignored for that particular element. A blank cell indicates that the condition does not apply to a cell, for example, a class does not have a transient modifier so it is blank.

TABLE 2-4 Signature File Content Summary

Modifier	Class or Interface	Field	Method	Constructor	Nested Class or Interface
`public`	+	+	+	+	+
`protected`		+	+	+	+
`abstract`	+		+		+
`static`		+	+		+
`final`	+	+	+		+

TABLE 2-4 Signature File Content Summary *(Continued)*

Modifier	Class or Interface	Field	Method	Constructor	Nested Class or Interface
`strictfp`	–		–		–
`transient`		–			
`volatile`		+			
`synchronized`			–		
`native`			–		

SignatureTest Command

The SignatureTest command has the following synopsis:

```
java com.sun.tdk.signaturetest.SignatureTest [arguments]
```

It follows the rules described in "Case Sensitivity of Command Arguments" on page 16. TABLE 2-5 lists the available arguments.

Note – SignatureTest command settings for the CLASSPATH environment variable and the -classpath argument are the same as those listed for the Setup command in TABLE 2-1.

Command Description

The SignatureTest command compares the reference API represented in a signature file to the API under test and produces a report. Depending on the command-line arguments specified, it uses either the reflection or static mode. If the -classpath argument is specified, the SignatureTest command checks if any extra classes are contained in the APIs it specifies.

See "Signature File Formats" on page 16 for details on supported versions of signature file formats.

TABLE 2-5 SignatureTest Command Arguments

Option	Description
`-help`	Optional. Displays usage information for available command arguments and exits.
`-debug`	Optional. Enables printing of the stack trace for debugging purposes if SignatureTest fails.
`-static`	Optional. Specifies to run in static mode without using reflection and reports on only the class files specified in the -classpath option.
`-mode [bin | src]`	Optional. Specifies the compatibility mode to use during the signature test, either binary or source, respectively. Defaults to src. "Source and Binary Compatibility Modes" on page 4 describes each mode.

TABLE 2-5 `SignatureTest` Command Arguments *(Continued)*

Option	Description
`-CheckValue`	Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.
`-NoCheckValue`	Specifies not to check the values of primitive and string constants.
`-ClassCacheSize` *size_of_cache*	Optional. Used in static mode only. Default is 1024. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function.
`-classpath` *path*	Optional. Specifies the path to one or more APIs to be tested. Defaults to the classes contained in the signature file under test. Can contain multiple directories or ZIP or JAR files. The `-package` argument filters the set of classes specified in `-classpath` (see "`CLASSPATH` and `-classpath` Settings" on page 10). Uses the path separator appropriate for the platform (identified by `java.io.File.pathSeparator`).
`-TestURL` *URL*	Optional. Specifies the directory location of the signature file as a `file` protocol URL: `file://`*path* Must end in a trailing forward slash on a UNIX system or a backslash on a Microsoft Windows or DOS system.
`-FileName` *file_name*	Required if `-Files` is not specified.. Specifies the name of a signature file to be used.
`-Files` *file_names*	Required if -FileName is not specified. Use this argument for testing a combination of APIs represented by corresponding signature files. Specifies the names of the signature files to be used delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See "Signature File `Merge` Rules" on page 11 for details on the rules used for merging.

TABLE 2-5 `SignatureTest` Command Arguments *(Continued)*

Option	Description
`-NoMerge`	Optional. Forces using the first encountered class description if more than one class with the same name is found in the input set of signature files specified by `-Files` option. In this case it uses the class from the left-most signature file that is specified with `-Files`. This option prevents the test from using the default merging behavior according to the JSR 68 rules. See "Signature File `Merge` Rules" on page 11 for details on the rules used for merging.
`-package` *package_or_class_name*	Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. The default is all classes and packages in the signature file. This argument acts as a filter on the set of classes or packages optionally specified in `-classpath`. Repeat the argument to specify multiple entries.
`-PackageWithoutSubpackages` *package*	Optional. Similar to the `-package` option, specifies a package to be reported on without its subpackages. Repeat the option to specify multiple entries.
`-exclude` *package_or_class_name*	Optional. A package or class to be excluded from the report, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-out` *file_name*	Optional. Prints report messages to a specified file instead of standard output.
`-FormatPlain`	Optional. Specifies not to sort report messages, but output them immediately. Allows a decrease in memory consumption compared to the default sorted format of message reporting.
`-apiVersion` *version_string*	Optional. Specifies the API version number of the implementation under test to be recorded in the report.
`-verbose`	Optional. Prints all error messages for inherited class members. Default is to remove all these error messages.
`-ErrorAll`	Optional. Specifies to make the signature test more strict by upgrading certain warnings to errors.

▼ Running a Signature Test With the JavaTest Harness

Although the `SignatureTest` command can be run by itself, `SignatureTest` can also be executed as a test by the JavaTest harness. This section describes the items that must be in place in a test suite to perform the signature test with the JavaTest harness. If you are developing an API, it is advisable to provide a signature test along with any behavioral tests you develope for that API. For information on how to create a test suite see the *JavaTest Architect's Guide* available at:

`http://java.sun.com/javame/reference/apis.jsp#javatest`

Also see Appendix A for an example of running the `SignatureTest` command from the command line without the JavaTest harness.

The following procedure describes how to add the signature test to your test suite for use with the JavaTest harness.

1. **Create the `.sig` file as described in "`Setup` Command" on page 14.**
2. **Create the following directory in your test suite and copy your `.sig` file to it:**

   test_suite-path`/tests/api/signaturetest`

   Where *test_suite-path* is the path to the base directory of the test suite installation.
3. **Use the `jar` utility to extract the classes from `sigtest.jar` into the `/classes` directory of your test suite, for example:**

   `%` cd *test_suite-path*`/tests/api/signaturetest/classes`

   `% jar -xvf` *path*`/sigtest.jar`

   The `sigtest.jar` file contains the classes necessary to run the signature test under the control of the JavaTest harness during a test run. You use the `sigtestdev.jar` only for signature test development purposes.
4. **Depending on which test description file you use, copy one of the test description files provided in the SigTest tool distribution to the `tests/` directory in your test suite.**

   The test description files are located in the following directory of the SigTest tool installation along with the other sample signature test files:

   SigTest-Directory`/examples/sampleTCK/tck/tests/api/sigtest/`

   Open these files now in your favorite text editor to review their contents.

   If you use the tag-based test description file, copy `SignatureTest.java` to the following location, but if you use the HTML test description file, copy `SignatureTest.html` to this location instead:

   testsuite-path`/tests/api/signaturetest/`

   Where *testsuite-path* is the path to the base of the test suite installation.

5. **Change the following two arguments in the `executeArgs` value of the test description file in use.**

- Change the argument to the `-FileName` option in the test description to correspond to the signature file name. It will look something like this:
  `-FileName ./`*name*`.sig`
- Change the argument to the `-package` option in the test description to the top-level package you are testing:
  `-package` *package_name*

When these items are in place, the JavaTest harness handles execution of the signature test as part of the test suite. It does so by executing the following class by specifying it in the test description file as the value of the `executeClass` parameter with arguments specified by the `executeArgs` parameter:
`com.sun.tdk.signaturetest.Test`.

Report Formats

You can cause `SignatureTest` command report messages to be sorted (default), or unsorted by specifying the `-FormatPlain` flag at the command line. See CODE EXAMPLE A-4 under "Example `SignatureTest` Command" on page 32 to see a sorted report that was generated into a plain text file with the `-out` option. Report messages contain the following types of information:

- The versions of both the reference API and the API under test
- The total number of errors found
- The differences such as added or missing classes, superclasses, fields, or methods
- The fully qualified name of the enclosing class related to any missing or added description
- Two errors for any modified item, one for a missing item and another for an added item
- A description of any incompatibility in the API implementation under test

Sorted Report

Report messages are sorted by default. Unlike the unsorted format, duplicate messages are removed. To accurately compare the error totals between formats, the sorted report prints two error counts: the total number of errors and the total number of duplicates.

The sorted report groups error messages by category of difference with classes within each category ordered alphabetically by name, and empty categories are ignored. This is the category ordering sequence:

- Missing Classes
- Missing Class Descriptions (Modified classes and nested classes)
- Missing Superclasses or Superinterfaces
- Missing Fields
- Missing Constructors
- Missing Methods
- Added Classes
- Added Class Descriptions (Modified classes and nested classes)
- Added Superclasses or Superinterfaces
- Added Fields
- Added Constructors
- Added Methods
- Linkage Errors

Linkage errors occur during testing if the API implementation under test is corrupted, for example, if a superclass or superinterface of a class under test cannot be loaded.

See the sorted report in CODE EXAMPLE A-4.

Unsorted Report

The `-FormatPlain` option specifies an unsorted report. The unsorted format reports messages immediately during execution and duplicate messages are included.

CODE EXAMPLE 2-1 shows an unsorted report that corresponds to the sorted report in CODE EXAMPLE A-4.

CODE EXAMPLE 2-1 Unsorted Report Example

```
Definition required but not found in example.test
    method public int get(int)
Definition found but not permitted in example.test
    method public java.lang.String get(int)
Definition found but not permitted in example.test
    method public void put()
SignatureTest report
Tested version: 2.0
STATUS: Failed. 3 errors
```

`SetupAndTest` Command

The `SetupAndTest` command has the following synopsis:

`java com.sun.tdk.signaturetest.SetupAndTest` [*arguments*]

TABLE 2-6 describes all arguments available to the `SetupAndTest` command. See "Case Sensitivity of Command Arguments" on page 16 and "`CLASSPATH` and `-classpath` Settings" on page 10. `SetupAndTest` requires `sigtestdev.jar` in the `CLASSPATH` environment variable.

Command Description

`SetupAndTest` is a wrapper command that runs only in the static mode to execute the `Setup` and `SignatureTest` commands in one operation. It performs these functions:

1. Calls the `Setup` command to create a signature file from the reference API specified as input. It creates a temporary signature file if no name is specified for it with the `-FileName` option.
2. Calls the `SignatureTest` command to make the comparison and produce a report.

TABLE 2-6 `SetupAndTest` Command Argument

`-help`	Optional. Displays usage information for available command arguments and exits.
`-reference` *path*	Required. Specifies the path to the classes to be used for creating the signature file.
`-test` *path*	Required. Path to the packages being tested.
`-FileName` *file*	Optional. Specifies the signature file name. The command execution uses a temporary file if not specified.
`-package` *package_or_class_name*	Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. This argument acts as a filter on the set of classes or packages that are tested and reported on. The default is all classes and packages. Repeat the argument to specify multiple entries.

TABLE 2-6 `SetupAndTest` Command Argument *(Continued)*

`-PackageWithoutSubpackages` *name*	Optional. Specifies package to be tested excluding subpackages.
`-exclude` *name*	Optional. A package or class to be excluded from testing, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-ClassCacheSize` *size_of_cache*	Optional. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function. Defaults to 100.
`-CheckValue`	Optional. Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.
`-NoCheckValue`	Optional. Specifies not to check the values of primitive and string constants.
`-verbose`	Optional. Enables error diagnostics for inherited class members.
`-apiVersion` *version_string*	Optional. Specifies the API version number of the implementation under test to be recorded in the generated report.
`-out` *file_name*	Optional. Prints report messages to a specified file instead of standard output.
`-FormatPlain`	Optional. Specifies not to sort report messages, but output them immediately. Allows a decrease in memory consumption compared to the default sorted format of message reporting.

`Merge` Command

The `Merge` command has the following synopsis:

`java com.sun.tdk.signaturetest.Merge` [*arguments*]

TABLE 2-7 describes the arguments available to the `Merge` command. Also see "Case Sensitivity of Command Arguments" on page 16.

Command Description

The Merge command combines (merges) a number of input signature files into one resulting signature file. See "Signature File `Merge` Rules" on page 11 for details on the rules used for merging, and "Signature File Formats" on page 16 for details on supported versions of signature file formats..

TABLE 2-7 `Merge` Command Arguments

Argument	Description
`-help`	Optional. Displays usage information for available command arguments and exits.
`-Files`	Required. Specifies the names of the signature files to be merged delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See "Signature File `Merge` Rules" on page 11 for details on the rules used for merging.
`-Write`	Required. Specifies the resulting output signature file.
`-Binary`	Optional. Specifies to use the binary merge mode. See "Source and Binary Compatibility Modes" on page 4.

APPENDIX **A**

Quick Start Examples

This appendix provides a simple example of each of the SigTest tool commands that you can run quickly to become familiar with them. It contains these sections:

- Example `Setup` Command
- Example `SignatureTest` Command
- Example `SetupAndTest` Command
- Merge Examples

Note – These examples are meant to be run in sequential order with the commands using output created by the previous example. The entire sequence can be completed in less than an hour if the required Java SE 5.0 platform runtime environment is already installed.

Example `Setup` Command

The following procedure illustrates using the `Setup` command to create a signature file.

Note – The example command lines in this appendix use the UNIX system syntax with a backspace character (\) to break long command lines.

▼ Running the `Setup` Command

1. **At a command prompt, change to a current working directory of your choice and note that all subsequent steps and commands are relative to this directory.**

2. **Using your favorite text editor, copy and paste the code from the `test.java` file in CODE EXAMPLE A-1 into a file with the same name under a new `V1.0` subdirectory, with this as a result:**

   current-working-dir`/V1.0/test.java`

   The example uses a compiled version of this file as a reference from which to create the signature file.

3. **Use the following command line to compile the `test.java` file into a target class file:**

   `% javac -d V1.0 V1.0/test.java`

   Where `javac` is a properly installed compiler that is compatible with the Java SE platform.

   This is the file that results from the command:

   `./V1.0/example/test.class`

   This class is used as the reference of comparison API in subsequent steps by creating a `test.sig` file to represent it.

4. **Create the `test.sig` signature file to represent class `test` by running the following `Setup` command.**

   This command-line example assumes you set the `CLASSPATH` environment variable to contain the `sigtestdev.jar` and the `JAVA_HOME` variable is set to the base directory of the Java runtime environment installation.

```
% java -cp V1.0:$CLASSPATH \
com.sun.tdk.signaturetest.Setup \
-classpath V1.0:$JAVA_HOME/jre/lib/rt.jar \
-static \
-apiVersion V1.0 \
-package example \
-FileName test.sig
```

The command produces a console message similar to the following:

```
Constant checking: on
Found in total: 12749 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

5. **Confirm the correct contents of the resulting `test.sig` file by comparing it with CODE EXAMPLE A-2.**

   This completes the `Setup` command example.

Note – Save the files you created in this procedure for use in the subsequent example in "Running `SignatureTest` Without the JavaTest Harness" on page 32."

CODE EXAMPLE A-1 The `V1.0/test.java` File

```
package example;
public class test {
    public <T> T get (T x) {
        return x;
    }
}
```

CODE EXAMPLE A-2 The `test.sig` File

```
#Signature file v4.0
#Version V1.0

CLSS public example.test
cons public test()
meth public <%0 extends java.lang.Object> {%%0} get({%%0})
supr java.lang.Object

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedException
meth public final void wait(long) throws java.lang.InterruptedException
meth public final void wait(long,int) throws java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()
```

Example `SignatureTest` Command

This section illustrates how to run `SignatureTest` from the command line without the JavaTest harness. Also see "Running a Signature Test With the JavaTest Harness" on page 23 to see how the signature test is run automatically during a test run.

▼ Running `SignatureTest` Without the JavaTest Harness

This procedure uses the same `test.sig` file (contents in CODE EXAMPLE A-2) and the same environment setup created in "Running the `Setup` Command" on page 30.

1. **If necessary, perform the procedure previously described in** "Running the `Setup` Command" on page 30.

   The procedure sets variables and creates the `test.sig` file that is used in subsequent steps.

2. **Using your favorite text editor, copy the code contents shown in** CODE EXAMPLE A-3 **into a second version of the** `test.java` **file and save it in a newly created** `V2.0` **subdirectory with this as the resulting path to the file:**

   current-working-dir`/V2.0/test.java`

CODE EXAMPLE A-3 The `V2.0/test.java` File

```
package example;

public class test {

    public String get (int x) {
        return "";
    }

    public void put () {
    }
}
```

3. **Compile the `V2.0/test.java` source file version with this command, noting that this newly compiled version serves as the implementation class under test.**

   `% javac -d V2.0 V2.0/test.java`

   This step generates the following new class file to be tested against the `test.sig` file in the next step:

   `./V2.0/example/test.class`

4. **Run the `SignatureTest` command against the new class file using the `-out` option to direct the results to a `report.txt` file with this command:**

```
% java -cp V2.0:$CLASSPATH \
com.sun.tdk.signaturetest.SignatureTest \
-apiVersion 2.0 \
-package example \
-FileName test.sig \
-out report.txt
```

   To set the `CLASSPATH`, see "`CLASSPATH` and `-classpath` Settings" on page 10. The command produces a console message similar to this:

   `STATUS:Failed.3  errors`

5. **Examine the result of the comparison made by the tool in the newly created `report.txt` file noting that it should be similar to CODE EXAMPLE A-4 with possible differences due to your system characteristics.**

   This report file lists the tested differences between the reference API and the implementation under test. Also see "Report Formats" on page 24.

Note – Save the files you created in this procedure for use next in "Example `SetupAndTest` Command" on page 34.

CODE EXAMPLE A-4 The `report.txt` File

```
SignatureTest report
Tested version: 2.0
Check mode: src [throws normalized]
Constant checking: off


Missing Methods
----------------

example.test:       method public <%0 extends java.lang.Object> {%%0}
example.test.get({%%0})
```

CODE EXAMPLE A-4 The `report.txt` File

```
Added Methods
-------------

example.test:       method public java.lang.String example.test.get(int)
example.test:       method public void example.test.put()


STATUS:Failed.3 errors
```

Example `SetupAndTest` Command

The following procedure runs the `SetupAndTest` command to create a new `test2.sig` file from the input API class, and compare it with the specified `example` package.

▼ Running the `SetupAndTest` Command

This procedure assumes the same current working directory as the previous example in "Running `SignatureTest` Without the JavaTest Harness" on page 32, and the same environment setup. It uses these files from previous examples:

- `./V1.0/example/test.class` - The API class used as the reference
- `./V2.0/example/test.class` - The API under test

1. **Set the `JRE` environment variable for access to the Java runtime environment files.**

   The `SetupAndTest` command running in static mode must have access to all superclasses and superinterfaces of any public classes that are under test, such as `java.lang.Object`.

   `% setenv JRE` *Java-Home*`/jre/lib/rt.jar`

   Where *Java-Home* is the base directory of the Java runtime environment installation.

2. **Run the following `SetupAndTest` command:**

```
% java $CLASSPATH com.sun.tdk.signaturetest.SetupAndTest \
-apiVersion 2.0 \
-reference V1.0:$JRE \
-test V2.0:$JRE \
-package example \
-FileName test2.sig
```

The previous `SetupAndTest` command reports to standard output a message similar to CODE EXAMPLE A-5 with differences according to your system characteristics

CODE EXAMPLE A-5 `SetupAndTest` Command Output Example

```
Invoke Setup ...
Class path: "V1.0;C:\java\jdk1.5.0_06\/jre/lib/rt.jar"
Constant checking: on
Found in total: 12749 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
Invoke SignatureTest ...
SignatureTest report
Tested version: 2.0
Check mode: src [throws normalized]
Constant checking: on


Missing Methods
---------------

example.test:      method public <%0 extends java.lang.Object> {%%0} example.te
st.get({%%0})

Added Methods
-------------

example.test:       method public java.lang.String example.test.get(int)
example.test:       method public void example.test.put()


STATUS:Failed.3 errors
```

Merge Examples

This section illustrates an example of running `Merge` which involves these three steps:

1. Compiling three `.java` files to produce `.class` files as a source for the signature files used in the example
2. Running the `Setup` command on each `.class` file to produce its signature file
3. Using `Merge` to combine the files and see the results.

▼ Running Merge Examples

These `Merge` examples use the environment variables listed in TABLE A-1, with example value settings and later command lines shown using UNIX system command syntax.

TABLE A-1 Environment Variable Settings for Merge Examples

Environment Variable	Description
`CLASSPATH`	Must include the `sigtestdev.jar` file, for example set to: `/sigtest-1.5/lib/sigtestdev.jar`.
`RT_JAR`	The location of the runtime `rt.jar` file , for example set to: `/opt/jdk1.5.0_09/jre/lib/rt.jar`.

1. **Set up the environment along with the environment variable settings in TABLE A-1.**

2. **Create each of the following files as listed in a separate directory but with the same file name:**

   Contents of: `./1/A.java`

```
package x;
public class A {
    public void abc() {}
    public void foo() {}
}
```

Contents of: `./2/A.java`

```
package x;
public class A {
    public void abc() {}
    public void bar() {}
}
```

Contents of: ./3/A.java

```
package x;
public class A {
    public static void abc() {}
}
```

3. **Run these commands to compile each file into a separate x subdirectory:**

```
% javac -d 1 1/A.java
% javac -d 2 2/A.java
% javac -d 3 3/A.java
```

These are the resulting files:

- ./1/x/A.class
- ./2/x/A.class
- ./3/x/A.class

4. **Run these three Setup command lines on each A.class file to produce the three x#.sig files as shown.**

   a. **Run command #1:**

```
% java -cp 1:$CLASSPATH com.sun.tdk.signaturetest.Setup \
-static -classpath 1:$RT_JAR -package x -FileName x1.sig
```

The command generates the ./x1.sig file shown in CODE EXAMPLE A-6 and produces a console message similar to this indicating successful setup with some possible differences according to your system characteristics:

```
Class path: "1:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

b. **Run command #2:**

```
% java -cp 2:$CLASSPATH com.sun.tdk.signaturetest.Setup \
-static -classpath 2:$RT_JAR -package x -FileName x2.sig
```

The command generates the `./x2.sig` file shown in CODE EXAMPLE A-7 and produces a console message similar to this indicating successful setup with some possible differences according to your system characteristics:

```
Class path: "2:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

c. **Run command #3:**

```
% java -cp 3:$CLASSPATH com.sun.tdk.signaturetest.Setup \
-static -classpath 3:$RT_JAR -package x -FileName x3.sig
```

The command generates the `./x3.sig` file shown in CODE EXAMPLE A-8 and produces a console message similar to this indicating successful setup with some possible differences according to your system characteristics:

```
Class path: "3:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

Now you can merge the signature files you just created.

5. **Run this command to merge** `x1.sig` **and** `x2.sig` **and produce the** `x1+x2.sig` **file shown in** CODE EXAMPLE A-9**:**

```
% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files
x1.sig:x2.sig -Write x1+x2.sig
```

The command generates a console message similar to this:

```
Warning: class java.lang.Throwable not found
STATUS:Passed.
```

6. **Run this command to merge `x2.sig` and `x3.sig` attempting to produce the `x2+x3.sig` file:**

```
% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files
x2.sig:x3.sig -Write x2+x3.sig
```

The command prints a message to the console similar to the following with possible differences due to system characteristics indicating a conflicting `static` modifier, and no signature file is created:

```
x.A.abc : <static> modifier conflict
STATUS:Error.Error
```

The conflict is that `x2.sig` contains this method (see CODE EXAMPLE A-7):

```
meth public void x.A.abc()
and x3.sig contains this method:
meth public static void x.A.abc()
```

Example Result Files

This section contains the files that are generated from the previous `Merge` examples.

CODE EXAMPLE A-6 Contents of `./x1.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()
```

CODE EXAMPLE A-6 Contents of `./x1.sig` *(Continued)*

```

CLSS public x.A
cons public A()
meth public void abc()
meth public void foo()
supr java.lang.Object
```

CODE EXAMPLE A-7 Contents of `./x2.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public void abc()
meth public void bar()
supr java.lang.Object
```

CODE EXAMPLE A-8 Contents of `./x3.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
```

CODE EXAMPLE A-8 Contents of `./x3.sig` *(Continued)*

```
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public static void abc()
supr java.lang.Object
```

CODE EXAMPLE A-9 Contents of `x1+x2.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public void abc()
```

CODE EXAMPLE A-9 Contents of `x1+x2.sig` *(Continued)*

```
meth public void bar()
meth public void foo()
supr java.lang.Object
```

Index